Feature-based Volumetric Terrain Generation and Decoration

Michael Becher, Michael Krone, Guido Reina, and Thomas Ertl

(Invited Paper)

Abstract—Two-dimensional height fields are the most common data structure used for storing and rendering of terrain in offline rendering and especially real-time computer graphics. By its very nature, a height field cannot store terrain structures with multiple vertical layers such as overhanging cliffs, caves, or arches. This restriction does not apply to volumetric data structures. However, the workflow of manual modelling and editing of volumetric terrain usually is tedious and very time-consuming. Therefore, we propose to use three-dimensional curve-based primitives to efficiently model prominent, large-scale terrain features. We present a technique for volumetric generation of a complete terrain surface from the sparse input data by means of diffusion-based algorithms. By combining an efficient, feature-based toolset with a volumetric terrain representation, the modelling workflow is accelerated and simplified while retaining the full artistic freedom of volumetric terrains. Feature Curves also contain material information that can be complemented with local details by using per-face texture mapping. All stages of our method are GPU-accelerated using compute shaders to ensure interactive editing of terrain. Please note that this paper is an extended version of our previously published work [1].

Index Terms—Terrain, interactive modeling, volumetric representations, spline curves, diffusion, GPU, surface texture, Ptex.

1 Introduction

The study and depiction of landscapes has a long-standing history in traditional arts such as painting (evidence suggests landscape paintings date back to as early as Ancient Greece [2]) and photography as well as in literature. It is no surprise that virtual landscapes have been present in the field of computer graphics since its early days. As early as 1980, animated short *Von Libre* by Loren Carpenter explored a mountain scene depicting fractal terrain [3]. In the real-time computer graphics of the 1970s and early 1980s, virtual landscapes were still mostly limited to basic silhouettes drawn by vector graphics or rather coarse 2D raster graphics. But with the rapid increase of processing power, more advanced computer graphics algorithms, and the advent of hardware-accelerated 3D computer graphics, the quality of virtual terrain improved likewise. Over the decades, many approaches to rendering were explored, including bitmap sprites and voxel graphics. Eventually, most real time applications settled on using 3D polygonal meshes to render detailed terrain surfaces. Behind the scenes, 2D heightfields (or heightmaps) were established as a common representation for terrain. To this date, heightmaps remain the standard solution for rendering terrain in some of the most popular, commercial game engines including Unreal Engine 4 [4], CryEngine V [5] and Unity [6]. However, overhanging terrain structures or caves simply cannot be represented by a heightmap without further extensions. Even terrain without overhangs, but with steep slopes, cliffs or any arbitrary terrain structures that stretch out vertically, can cause issues due to stretched-out textures or quads and triangles with high aspect ratio. Such restrictions do not apply to terrain representations in a three-dimensional domain. It is possible to represent and store any terrain structure, including in particular overhanging structures, caves or generally terrain with several vertical layers, in a volumetric data structure. Corresponding research, frameworks or extensions to existing graphics engines are indeed available. However, manual creation and editing of such volumetric terrain representations often becomes a tedious task if every cubic meter has to be manually *painted* with a brush (or rather blob) tool. In many cases it is preferred to generate volumetric terrain data by procedural algorithms. This, on the other hand, limits the control the user can exercise over specific regions of the terrain.

We present an efficient and intuitive method for controlled generation and fast editing of volumetric terrain. We propose to apply concepts and modelling tools that have previously been used to create terrain heightmaps to a volumetric representation. The immediate intention is to let users model prominent, large-scale terrain features that define the overall appearance of the landscape. Inspired by the use of curve-based tools for heightmap creation, Feature Curves, i.e. parametric 3D spline curves, are used to model the terrain and place constraints in the scene (see Figure 1). Editing of large-scale terrain features is thereby reduced to adjusting a small amount of curve control vertices and constraint properties. We developed a multi-stage GPU-accelerated computational pipeline that generates a volumetric terrain representation based on the Feature Curves. The pipeline works on a specified domain containing a set of discrete vector and scalar fields. Voxelization is used to transfer the (sparse) information defined by the Feature Curves into these fields. Using a diffusion process, a dense normal vector field is obtained and then used to approximate a signed distance field for a terrain surface that matches the constraints given by the Feature Curves.The terrain surface

• *The authors are with the Visualization Research Center of the University of Stuttgart (VISUS), Germany.*
E-mail: {firstname}.{lastname}@visus.uni-stuttgart.de

Manuscript received April 19, 2005; revised August 26, 2015.

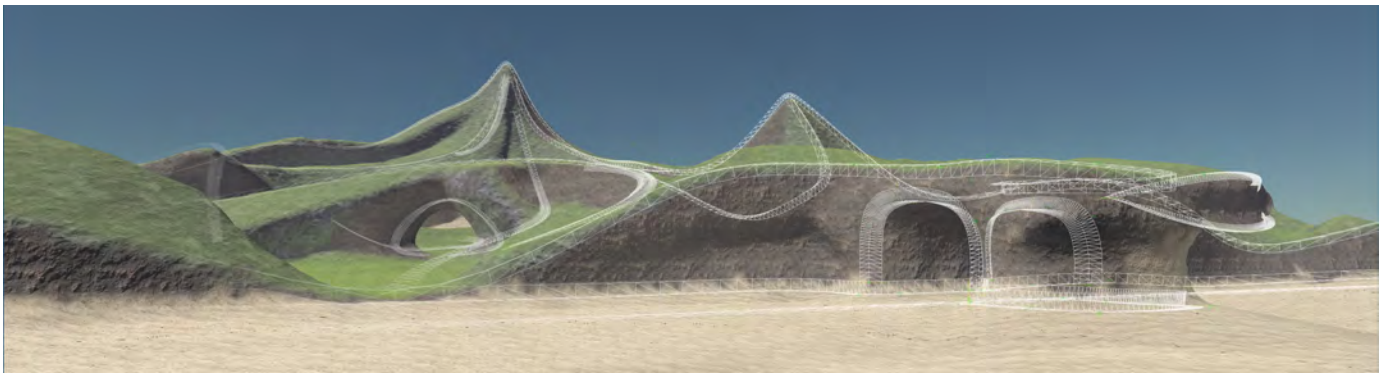

Fig. 1. A coastline modelled with our method. The Feature Curves used to model the scene are blended over the image taken from our modelling application. Note the overhanging rock faces and the arch in the front, which are not possible using a traditional two-dimensional heightfield.

is then extracted and, based on user-controlled parameters, medium to high resolution details are added by simple, procedural means to create a visually pleasing result. The Feature Curves also control the surface material and we use per-face texture mapping for rendering to be able to include locally unique details to the surface of the dynamically generated terrain mesh.

Our main contribution is a novel method for generating a terrain surface from a limited set of prominent terrain features with the following notable properties:

- Fast and intuitive modelling of terrain features by means of parametric curves;
- No restrictions to the vertical layout of the terrain in general (allowing overhanging terrain and caves) by using a volumetric computational domain;
- Interactive, real-time editing of terrain
- Allows importing heightmaps and pre-modelled terrain assets and combining them with the (voluemtric) terrain features
- Efficient texturing using per-face texture mapping

Please note that this paper is an extended version of our work previously presented at the 2017 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (i3D) [1].

The rest of the paper is structured as follows. After a brief overview of related work in the field of terrain modelling in Section 2, we introduce our modelling primitives in Section 3. In Section 4, we present our multi-stage computational pipeline to generate arbitrary terrain surfaces and discuss each stage in detail. Extensions of our method and an application prototype are presented in Section 5 and 6, respectively. In Section 7 we discuss the results of our method with regard to performance, capabilities, and quality. Section 8 concludes the paper and discusses possible future extensions.

2 Related Work

There are many efficient methods for modelling and editing terrain that range from fully procedural methods to methods combining sparse or dense user input with an automated terrain generation process. An overview of different methods used to model and represent terrain is given in the recent work of Natali *et al.* [7]. Different techniques are classified into data-oriented scenarios and workflow-oriented ones and a comparison of their abilities is made. With regard to the classification of Natali *et al.*, the method we present is a workflow-oriented, data-free technique.

Procedural methods are a popular approach to generating terrain surfaces as they generally require little input to create realistic landscapes. A comprehensive overview of general procedural approaches in computer graphics was given by Ebert *et al.* [8]. A survey focusing on methods used to procedurally generate terrain was given by Smelik *et al.* [9]. Some procedural approaches that are specifically aimed at terrain generation model real-world effects, such as hydraulic erosion, that have a large impact on the appearance of landscapes [10]. Génevaux *et al.* presented a method based on hydrology that merges procedurally generated terrain with a user-defined river network [11].

Methods combining sparse user input with an automatic, and often procedural, terrain generation are of particular interest for our work. Sketch-based methods allow the user to sketch complex silhouettes and then generate a terrain surface that matches the input sketches [12] [13]. While conceptually similar to the idea of our methods, the sketches are usually made in a first person view whereas our method works in world space and furthermore also defines the slope around a terrain feature. Hnaidi *et al.* presented a framework for modelling terrain based on parametric spline curves [14]. The *Feature Curve* concept they introduced for modelling terrain features has been a notable inspiration for this paper. However, in contrast to our method, the technique they presented is used to create heightmaps. Emilien *et al.* focus on the modelling of waterfalls and combine procedural methods with user-controlled tools [15]. Although also using heightmaps, they achieve overhanging terrain by vertical vertex displacement. A framework with volumetric terrain generation in mind has been presented by Peytavie *et al.* [16]. They combine material layers stored in a volumetric datastructure with an implicit surface representation and offer high level tools for sculpting the terrain surface.

A more generic approach to modelling arbitrary objects using parametric curves was presented by Singh and Fiume [17]. Like our method, they propose to manipulate curves to implicitly modify the surface of objects rather than to directly manipulate the surface representation itself. However Singh and Fiume use curves to deform existing geometry of objects, whereas our method generates completely new geometry that satisfies the constraints given by our curve-

based modelling primitives. Adding and manipulating additional curves will not deform the previously generated geometry but will simply cause a complete regeneration of the geometry with a potentially different topology.

3 Modelling Primitives

Rather than directly shaping and modifying parts of the terrain surface (e.g. by using brush-based tools), we propose to place primitives in the scene that describe and constrain the terrain surface in their vicinity and from which the terrain surface is propagated into the whole domain. While this is a relatively straightforward concept, we introduce some novelty by applying it to an otherwise unrestricted three-dimensional computational domain.

Our primary modelling primitive are Feature Curves, a concept previously used in the context of terrain modelling by Hnaidi *et al.* [14]. The term is derived from the idea that curve-based modelling primitives are used to describe prominent terrain features such as ridgelines, river beds, or cliffs. Due to the shape of many terrain features, like the aforementioned ones, parametric curves are a good fit to design a generic modelling tool for describing such features. Our concept of a Feature Curve shares the same basic ideas, the exact definition and properties however differ in several aspects from those of Hnaidi *et al.*. Most prominently, we define our Feature Curves in three-dimensional space and are thus able to describe almost arbitrary terrain surface structures including, in particular, structures that are impossible to describe with a heightfield as they are compromised of several vertical layers.

3.1 Feature Curves

Our Feature Curves are constructed by augmenting regular, three-dimensional, cubic B-Splines with additional attributes. These attributes are gathered in so-called *Constraint Points* that are placed on and associated with – we also say attached to – a specific Feature Curve. A Constraint Point p attached to a Feature Curve $\mathcal{F}$ is defined as a tuple $\left(b_l, b_r, \alpha, \beta, m, t\right)$ where $b_l, b_r \in \mathbb{R}^3$ are the left- and right-hand bitangent vector, both α and β are scalar values used for noise generation, m is a surface material ID, and the scalar value t denotes the position of the Constraint Point on $\mathcal{F}$ in the curve's parameter space. The vectors b_l and b_r are perpendicular to the curve's tangent vector at the curve point given by t and control the slope of the terrain surface. Formally, a Feature Curve is then defined as

$$\mathcal{F}(t) = \left(\mathbf{B}(t), \mathcal{P}(t), s\right)$$

where $\mathbf{B}$ is a cubic B-spline, $\mathcal{P}$ is a parametric function that interpolates between constraint points in a given set P, and s is a boolean value that determines whether $\mathcal{F}$ acts as a seed primitive for the terrain surface or simply as a guidance primitive. Thus, a point on a Feature Curve is a tuple of a position in 3D space (given by the evaluation of a B-spline), a tuple of interpolated constraint attributes and the seed property of the Feature Curve.

A Feature Curve requires a minimum of two Constraint Points attached to the curve's endpoints, with an arbitrary number of additional Constraint Points added in between.

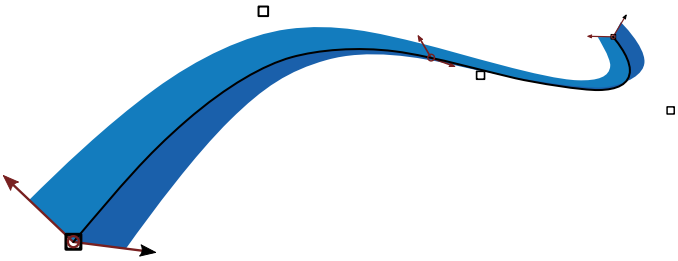

Fig. 2. A visual representation of a Feature Curve with five control vertices (black boxes) and three Constraint Points (red circles). At each Constraint Point, the bitangent vectors are displayed. Two ribbons matching the bitangent constraints are traced along the curve.

Except for the bitangent vectors and material IDs, the attributes given by the Constraint Points are linearly interpolated along the curve. Due to the curve's arbitrary shape and orientation in 3D space, the interpolation of bitangent vectors is non-trivial. A consistent interpolation is possible by first computing a rotational transformation matrix from known curve tangent vectors and then applying this matrix to the bitangent vectors. For the material ID, nearest neighbour interpolation is used along Feature Curves.

A pair of ribbons joined at one edge as seen in Figure 2 serves as the visual representation of a Feature Curve. Each ribbon can be thought of as the result of tracing the left- or right-hand bitangent vector along the curve by interpolating them between Constraint Points.

3.2 Proxy Geometry

So far, Feature Curves are defined as continuous parametric curves. However, with respect to the terrain generation process presented in the upcoming section, a representation that can be evaluated fast and conveniently in object space is desired. Furthermore we aim to find a representation that could be shared by different kinds of additional modelling primitives. Since modelling primitives are entities placed and arranged in the scene, proxy geometry is useful to give visual feedback of their placement and shape, especially the shape of the ribbons as depicted in Figure 2. Therefore, we decided to employ a triangle mesh as input for the terrain generation pipeline because it is most convenient to handle as shared representation both for rendering and for evaluation in the pipeline. Recall the visual representation of a Feature Curve as a pair of connected ribbons. To create a triangle proxy mesh for Feature Curves as depicted in Figure 2, Feature Curves are first discretized into piecewise linear segments. For each line segment, the left- and right-hand bitangent vector is interpolated at the endpoints and used to create two possibly distorted rectangles, that extend away from the line segment. Curve position and shape is implicitly stored in the vertex positions of the proxy mesh. The remaining constraint properties are stored as additional vertex attributes alongside a per-vertex normal vector which is derived from the curve tangent at the segment's endpoints and the interpolated bitangent vectors.

3.3 Heightmaps and Meshes

Although our method specifically aims to recreate terrain features that cannot be represented by 2D heightmaps, heig-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2017.2762304, IEEE Transactions on Visualization and Computer Graphics

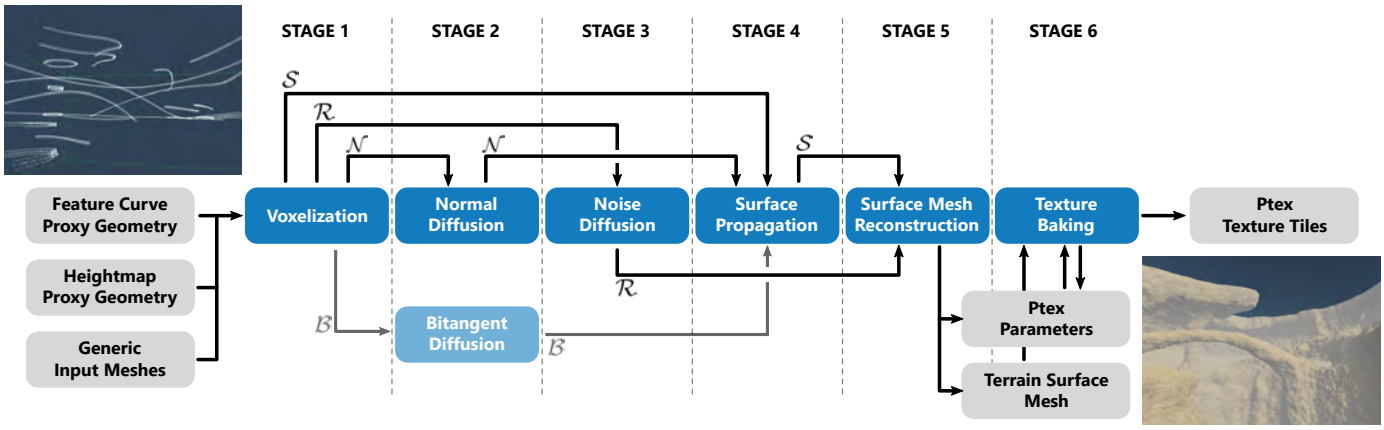

Fig. 3. The terrain generation pipeline as implemented in our method. Gray boxes depict in- and output of the pipeline. Blue boxes depict stages as described in Section 4.1. Arrows indicate data flow, i.e. fields, between stages.

htmaps and pre-modelled assets stored as triangle meshes still remain the most common terrain representation and, thus, a valuable input format. Therefore, in addition to the Feature Curves, we can also use heightmaps and arbitrary meshes as input primitives.

A heightmap can serve as a base-layer for a landscape that can subsequently be decorated using Feature Curves to incorporate any kind of terrain features that slope beyond the vertical. After importing the heightmap data, a proxy mesh is generated internally to comply with our generalised input format. The thus generated proxy meshes or imported meshes can be used as secondary modelling primitives. Using imported meshes allows us to include legacy content created by traditional terrain generation tools as long as these tools support mesh output.

4 Volumetric Terrain Generation Pipeline

The workflow we propose lets the user arrange a set of high-level modelling primitives in the scene from which a terrain surface mesh and material information for texturing is automatically generated. This is handled by a multi-stage GPU-accelerated computational pipeline that is described in detail in this section (see also Figure 3).

4.1 Overview

As one of the key goals of our method is to create terrain surfaces with several vertical layers, the terrain generation pipeline operates on a volumetric computational domain. All modelling primitives within this domain are processed by the pipeline as input. These modelling primitives implicitly define a surface, which in the following we refer to as the target terrain surface. In order to retrieve an explicit representation of this target surface, we make the basic assumption that it is predominantly smooth, that is the second derivative of its normal vectors equals zero. Furthermore, recall that by definition of the modelling primitives, the target terrain surface passes through the location of surface modelling primitives and in the vicinity of both surface modelling primitives and guidance ones, the surface complies to the normal vectors given by the nearest modelling primitives. From these properties it ensues that one can compute

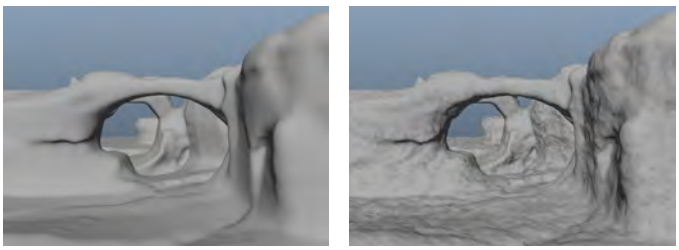

Fig. 4. A comparison between the terrain geometry before applying surface displacement based on the noise parameters stored in $\mathcal{R}$ (left) and afterwards (right).

a dense normal vector field using the normal vectors given by the modelling primitives as initial values for a diffusion process. Since this vector field conforms to the target surface properties (i.e. at the location of modelling primitives it is equal to the given surface normals and smooth everywhere else), it is also an implicit description and most notably a dense one for any potential terrain target surfaces. By combining it with information on points in the domain that are known to be located on the target terrain surface, an explicit representation of this target surface is computed.

The terrain surface generation is restricted to a limited domain that is defined by a discrete, three-dimensional, uniform, regular grid. All stages of the pipeline operate on a set of discrete vector- and scalar fields aligned to that grid. To store the information computed during the pipeline execution, the following fields are defined within the computational domain:

- $\mathcal{N}$: A vector field containing surface normal vectors
- $\mathcal{B}$: A vector field containing surface bitangent vectors
- $\mathcal{S}$: A scalar field storing terrain surface information
- $\mathcal{R}$: A vector field storing amplitude and roughness parameters for a noise generator, as well as a surface material ID

For the core terrain generation method, the normal vector field $\mathcal{N}$ is sufficient, while the bitangent vector field $\mathcal{B}$ is only required by an optional stage and an alternative algorithm. We also refer to $\mathcal{N}$ and $\mathcal{B}$ as guidance fields. The scalar field $\mathcal{S}$ stores a distance field that serves as an intermediate terrain representation. The parameters stored

in $\mathcal{R}$ are used to generate a noise field that we use to add additional high-frequency detail to the generated terrain surface (see Figure 4). Since storing the high-frequency noise signal would require are very high resolution, only noise generation parameters are stored in $\mathcal{R}$. The material ID stored alongside the noise parameters is used to texture the terrain according to the constraints given by the Feature Curves. The complete process of generating a terrain mesh from the sparse user input is split into the following six (plus one optional) stages, also shown in Figure 3.

1) **Voxelization** – Input modelling primitives are converted to the computational domain.
2) **Guidance field diffusion** – A dense normal vector field (and optionally a bitangent one) is computed using diffusion.

2a) **Feature Curve expansion** – *Optional.* The user-defined Feature Curves are automatically expanded along the guidance fields.

3) **Noise field diffusion** – A dense field of noise parameters is computed using a diffusion algorithm.
4) **Surface propagation** – Surface field values are propagated along the guidance field.
5) **Surface mesh reconstruction** – An iso-surface is extracted from the final surface field as a quad mesh.
6) **Texture Baking** – Texture memory and layouts are prepared and vista texture tiles for all quads are baked.

4.2 Voxelization

In the initial stage of the terrain generation pipeline, the input modelling primitives are converted from their vector-based definition in continuous 3D space to the discrete computational domain. These steps are shown in Figure 5. Simply speaking, we use the triangle mesh proxy geometry generated from the parametric curves as input for the voxelization algorithm instead of the parametric representation itself. That way, the input of the pipeline is generalised to triangle meshes, which allows to not only to process the proxy meshes of Feature Curves but also generic input meshes, as for example a pre-modelled asset containing a section of terrain, or a proxy mesh generated from a heightmap.

The proxy meshes of all relevant modelling primitives have to be discretized into a three-dimensional, regular, uniform grid, a process that is often referred to as voxelization. We chose a generic approach and simply determine all intersected grid cells for each triangle using a geometric intersection test. Since a grid cell can be intersected by multiple triangles with different normal and bitangent vectors as well as different noise generation parameters, the values of all intersecting triangles are first simply gathered per grid cell.

Gathering the triangles of Feature Curve proxy meshes, generic input meshes and heightmap proxy meshes is split into three separate passes. While this also simplifies the handling of different vertex formats, it primarily allows us to limit the voxelization of input mesh triangles to regions where no Feature Curve is present or close, and the voxelization of heightmap proxy mesh triangles to regions where neither Feature Curves nor input meshes are close. Defining an explicit prioritization of Feature Curves over input meshes and heightmaps and also of input meshes over heightmaps has two advantages: It is necessary in order to smoothly merge surfaces generated by different inputs. Furthermore, it would otherwise not be possible to use Feature Curves to decorate existing terrain with the additional features they can describe (i. e. to carve holes, trenches and tunnel openings into the terrain surface generated by a heightmap or input meshes). The significance of discarding triangles from inputs with lower priority for the interaction between the different modelling primitives is illustrated in Figure 6. In the gathering step of the voxelization stage, Feature Curve proxy mesh triangles are therefore processed first. In the second pass, the information of input mesh triangles is then only gathered for intersected voxels if no voxel in the vicinity was previously already intersected by a Feature Curve triangle. Finally, in the third pass, heightmap proxy mesh triangles are only considered if no voxel in the vicinity was previously intersected by either a Feature Curve or input mesh triangle. The radius around Feature Curves (and input meshes) for discarding mesh triangles from inputs with lower priority generally depends on the desired strength of the effect and the size of the voxels. Therefore, the radius of the vicinity is a user-defined parameter in object space, from which the number of voxels is computed.

Once all triangles are processed, the gathered values are averaged per voxel and written to the fields $\mathcal{N}$, $\mathcal{B}$ and $\mathcal{R}$ respectively. Note that if a cell is intersected by normals from both ribbons of a Feature Curve, normals derived from both bitangent vectors are averaged. The normal field is therefore smoothed along the spine of a Feature Curve. Special attention is required to average bitangent vectors. All left-hand and right-hand bitangent vectors in the cell are first averaged separately, resulting in two bitangent vectors $\hat{b}_l$ and $\hat{b}_r$. Then the vector b_{avg} that maximizes $|\hat{b}_l \cdot b_{avg}| + |\hat{b}_r \cdot b_{avg}|$ is computed. To that end, the averages are either added up if the angle between the two averaged vectors is smaller than $90°$, or subtracted otherwise. Furthermore, note that input meshes and heightmaps can typically not be expected to contain noise generation parameters or a material ID and we therefore globally assign user-defined values for those to the vertices of imported meshes and heightmap proxy geometry to avoid unnaturally smooth surfaces (see Figure 4). An alternative would be to import surface displacement values or noise parameters along with the mesh or heightmap data, e.g. a surface displacement texture map. However, the detail level will be limited by the resolution of our computational grid unless procedural parameters, as used by our method, are extracted.

For the upcoming sections, let $\Omega_N \subset \mathcal{N}$ and $\Omega_B \subset \mathcal{B}$ be the subset of voxels that have normal and bitangent vector information stored during the voxelization, $\Omega_R \subset \mathcal{R}$ the subset of voxels that have noise parameters stored and $\Omega_S \subset \mathcal{S}$ be the subset of voxels corresponding to cells that are intersected by at least a single triangle of a surface modelling primitive.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2017.2762304, IEEE Transactions on Visualization and Computer Graphics

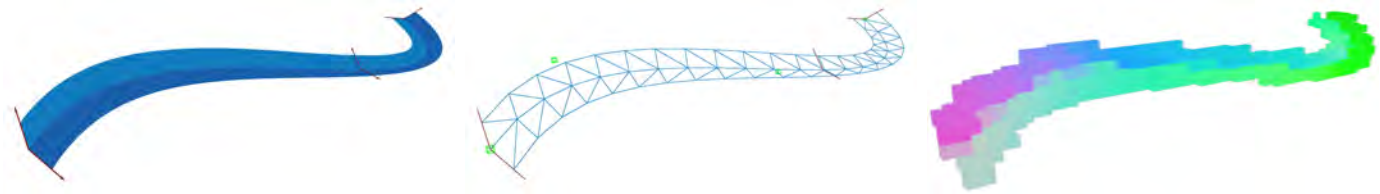

Fig. 5. Voxelization of a Feature Curve. Left: Illustration of the initial parametric Feature Curve. Middle: Feature Curve discretized to a proxy mesh. Right: Result of voxelizing the Feature Curve proxy mesh (voxels from field $\mathcal{N}$ are shown).

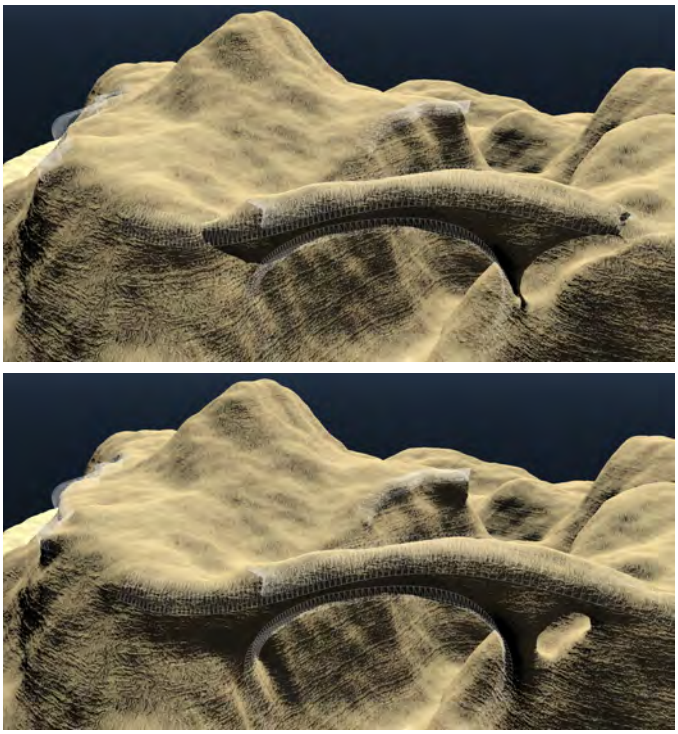

Fig. 6. Comparison between naïve voxelization of all input primitives and voxelization with a dead zone around Feature Curves. Top: The closed surface generated from a heightmap input cuts off the geometry generated from Feature Curves. Bottom: The surfaces generated by Feature Curves and heightmap input smoothly merge to a single terrain surface.

4.3 Normal Field Diffusion

After the voxelization stage, the normal vector field $\mathcal{N}$ is only sparsely populated. In order to use it for computing a closed terrain surface, it has to be densely populated. Recall that we require the target terrain surface to comply to the constraints given at Feature Curve locations and assume it to be smooth everywhere else. These properties should be reflected by the normal vector field. The problem of computing a smooth and densely populated vector field from sparse input shares similarities to gradient vector flow, a method for computing a dense vector field based on image gradient by minimizing an energy functional [18]. However, when applied to our problem, the method can be simplified to applying the smoothness term of the energy functional to $\mathcal{N} \setminus \Omega_N$. It follows that one can obtain a suitable dense normal vector field by solving a diffusion equation on the sparsely populated field $\mathcal{N}$ using the voxels in Ω_N as initial boundary conditions. To retain edges on the surface to at least some extent, we opt for using anisotropic diffusion. To that end, the following discretized anisotropic diffusion equation, as presented by Perona and Malik [19], is applied to $\mathcal{N} \setminus \Omega_N$ in an iterative computation:

$$\begin{aligned}\mathcal{N}^{t+1}_{i,j,k} = \mathcal{N}^{t}_{i,j,k} + \frac{1}{6}\Bigg(& c_E \cdot \left[\frac{\partial \mathcal{N}^t}{\partial x}\right]^+_{i,j,k} + c_W \cdot \left[\frac{\partial \mathcal{N}^t}{\partial x}\right]^-_{i,j,k} \\ &+ c_U \cdot \left[\frac{\partial \mathcal{N}^t}{\partial y}\right]^+_{i,j,k} + c_D \cdot \left[\frac{\partial \mathcal{N}^t}{\partial y}\right]^-_{i,j,k} \\ &+ c_N \cdot \left[\frac{\partial \mathcal{N}^t}{\partial z}\right]^+_{i,j,k} + c_S \cdot \left[\frac{\partial \mathcal{N}^t}{\partial z}\right]^-_{i,j,k}\Bigg),\end{aligned}$$

where $\left[\frac{\partial}{\partial x}\right]^+_{i,j,k}$ denotes the finite difference operator using a forward difference in x-direction at the location (i,j,k), while $\left[\frac{\partial}{\partial x}\right]^-_{i,j,k}$ denotes the backward difference operator. To evaluate the finite differences at the boundaries of the domain, homogeneous Neumann boundary conditions are used for $\mathcal{N}$. We define the coefficients c_E, c_W, c_U, c_D, c_N and c_S as a function g of the negative dot product of adjacent voxels in $\mathcal{N}$mapped to $[0,1]$, e.g.:

$$\begin{aligned} c_E &= g\left(\frac{1 - \left((\mathcal{N}^t_{i,j,k}) \cdot \mathcal{N}^t_{i+1,j,k}\right)}{2}\right), \\ c_W &= g\left(\frac{1 - \left((\mathcal{N}^t_{i,j,k}) \cdot \mathcal{N}^t_{i-1,j,k}\right)}{2}\right), \\ c_U &= \ldots \end{aligned}$$

The coefficients c_U, c_D, c_N and c_S are defined analogously for the two remaining coordinate axes. We choose function g as a sub-quadratic function known from literature [19]:

$$g(x) = \frac{1}{1 + (\frac{x}{k})^2}$$

The choice of parameter k depends on the desired amount of smoothing across edges and is empirically set to a fixed value of 0.01. While we found this to produce good results, this parameter could be added to the parameter set of Constraint Points to give additional control to the user. Note that for all voxels that have not yet been set to a meaningful value, i.e. $\mathcal{N}_{i,j,k} = \vec{0}$, the function g will evaluate to 1 due to the dot product being 0. In this case, the anisotropic diffusion equation simplifies to Laplace's equation [20]. We deliberately use this property in order to propagate normal vectors to regions of the field where no meaningful coefficient for the anisotropic diffusion can be computed yet.

4.4 Noise Field Diffusion and Material Propagation

Like the guidance vector fields, the field $\mathcal{R}$ is only sparsely populated after the voxelization and the initial values have to be propagated into the whole domain. As we assume the noise parameter component of the field to be smooth as well, we again use a diffusion process. However, we do not require the diffusion process in $\mathcal{R}$ to be edge-preserving and therefore simply solve Laplace's equation to obtain a dense field. The material ID, on the other hand, cannot be interpolated and smoothly distributed within the field as it is not a numerical but a categorical value. Instead, we propagate the material ID using a confidence value based on a cell's local neighbourhood. To that end, a fourth component of $\mathcal{R}$ is used to store a scalar value that expresses the confidence of a material assignment, which is set to 1.0 in Ω_R. During the iterative noise parameter diffusion, a material is assigned to each cell that has at least one adjacent cell with an already assigned material. The confidence is then averaged over the local neighborhood: given a cell c, the confidence values of all adjacent cells that have a material assigned are summed up per material. The material with the largest sum is then assigned to c and this sum divided by the total sum of values is assigned as the confidence for c.

4.5 Surface Propagation

Once $\mathcal{N}$ is computed, we can proceed with determining the surface field $\mathcal{S}$. $\mathcal{S}$ shall result from a signed distance function with regard to the target terrain surface. Obviously, at this point we do not have a surface as input to derive $\mathcal{S}$ but rather we want to compute $\mathcal{S}$ by different means in order to reconstruct the target surface from it. The basic idea is to propagate the surface information within $\mathcal{S}$ starting from the values stored in Ω_S during the voxelization stage. To compute $\mathcal{S}$ based only on the initial values in Ω_S and the guidance field $\mathcal{N}$, we define a function that approximates a voxel's distance value based on its local neighbourhood.

Let $\mathcal{NB}_{i,j,k}$ be the set of coordinates adjacent to the location (i,j,k) in the domain, i.e. $(i \pm 1, j, k)$, $(i, j \pm 1, k)$ and $(i, j, k \pm 1)$. Given a cell c_x with $x \in \mathcal{NB}_{i,j,k}$, assume that a target surface intersecting c_x is locally planar and its orientation is given by $\mathcal{N}_x$. Then we can model this surface as a plane defined by the midpoint of c_x and the normal vector $\mathcal{N}_x$. The signed distance d between the midpoint of cell $c_{i,j,k}$ and this plane is obtained by a simple point-plane distance calculation. In case the target terrain surface actually intersects c_x, i.e. $\mathcal{S}_x = 0$, the obtained distance value d is assigned to $\mathcal{S}_{i,j,k}$ unless a smaller distance is computed for one of the other neighbour cells. If, on the other hand, c_x is not intersected by the target surface but the distance to the nearest surface point is already known and stored in $\mathcal{S}_x$ we simply assign the sum $\mathcal{S}_x + d$ to $\mathcal{S}_{i,j,k}$. If c_x is neither intersected by the surface nor a valid distance is known yet, no value based on c_x is assigned to $\mathcal{S}_{i,j,k}$ at this point. Due to the construction of $\mathcal{N}$, this is a legitimate approximation to the distance value at location (i,j,k). The surface normal points in the direction of the nearest surface and is therefore a reasonable indicator for determining whether a voxel is closer or further away from a surface than the reference voxel. Recall that $\Omega_S \subset \mathcal{S}$ is the subset of $\mathcal{S}$ that is filled during the voxelization. Let $\partial\Omega_S \subset \mathcal{S} \setminus \Omega_S$ be the boundary between Ω_S and $\mathcal{S} \setminus \Omega_S$. Every value $\mathcal{S}_{i,j,k} \in \partial\Omega$ is set according to the following function and inserted into Ω_S.

$$\mathcal{S}_{i,j,k} = \min_{x \in \mathcal{NB}_{i,j,k}} \left(\mathcal{S}_x + ((i,j,k) - x) \cdot \frac{\mathcal{N}_{i,j,k} + \mathcal{N}_x}{2} \right)$$

Once $\Omega_S = \mathcal{S}$, a signed distance field for our target surface has been successfully estimated. Note that the actual calculation uses an interpolated normal located between (i,j,k) and x in order to reduce the error introduced by assuming the surface to be locally planar. Each value $\mathcal{S}_{i,j,k}$ is set only once. An iterative computation that tries to improve the distance approximation over time proved to be unstable, as modelling the surface contour as locally planar introduces an error that can lead to two neighbouring voxels continually de- or increasing each other's value. Further research into a more elaborate model of the local surface contour might improve this behaviour and the resulting distance approximation.

Because of discontinuities in $\mathcal{N}$as well as the discrete nature of $\mathcal{N}$, the surface propagation tends to suffer from noise as it assumes the target surface to be locally planar. By applying a Gaussian filter to $\mathcal{S}$, the noise is suppressed to a certain degree before $\mathcal{S}$ is used as input for the final pipeline stage.

As a side note, we also explored an alternative approach to propagating the surface information in $\mathcal{S}$. Here, a curve is traced through the guidance vector field similar to a streamline. If this streamline originates from a point known to be on the target surface, we assume that every voxel intersecting the streamline also intersects the target surface. Ultimately this approach proved to be less reliable, partly because it uses the biased bitangent vector field $\mathcal{B}$.

4.6 Surface Mesh Reconstruction

In the fifth stage of the pipeline, an explicit surface representation for efficient rendering is reconstructed from the implicit representation given by $\mathcal{S}$. We deliberately use polygonal meshes as these are efficient to render on GPUs and are supported by most rendering pipelines. The reconstruction of a surface mesh from $\mathcal{S}$ is equivalent to computing an iso-surface of the scalar field. As our method computes $\mathcal{S}$ as a signed distance field, the iso-value is set to zero. Note that in-between two Feature Curves that have a similar normal direction and are located above one another (*w.r.t.* their normal vectors), a sign change in $\mathcal{S}$ occurs and an iso-surface is extracted. This behaviour supports the generation of a plausible surface in areas not explicitly defined by modelling primitives. For example, if the roof of a cave is not explicitly modelled, it will be implicitly created where the positive distance values propagated from the cave floor meet the negative distance values propagated from the terrain surface above the cave.

Unlike our previous work, where we used the well-known Marching Cubes algorithm [21] to extract the iso-surfaces as a triangle mesh, we switched to the Naïve Surface Nets algorithm [22] to generate a quad mesh. For each cell that is intersected by the iso-surface, a vertex is generated and positioned within the cell based on the iso-values of the cell corners. Also, for each cell, quads are

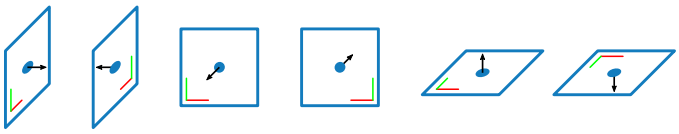

Fig. 7. The six quad configurations of the Naïve Surface Nets algorithm and the local uv spaces used in our implementation.

generated that connect to vertices of directly adjacent cells along the positive coordinate axes. The number of possibly generated quads per cell is therefore limited to three and there is a total of six different cases of generated quads. All six cases are also assigned a specific local uv-space as shown in Figure 7. The resulting mesh is a much better fit for the per-face texture mapping that we use for texturing the terrain surface.

During this stage, the noise parameters and material ID stored in $\mathcal{R}$ are also embedded into the vertex data of the extracted mesh, so that we can compute a corresponding high-frequency noise signal during rendering. We apply this noise as vertex displacement to obtain additional medium-scale details on the surface mesh. Furthermore, surface normal vectors that are required for lighting and procedural texturing of the extracted mesh are also computed and embedded in the vertex data. To obtain normal vectors that consistently match the extracted surface mesh, we compute the normals as the gradient of $\mathcal{S}$ instead of using the values stored in $\mathcal{N}$.

Using per-face texture mapping (Ptex) requires additional per-primitive parameters to access texture data of adjacent primitives on the boundary between primitives. A GPU buffer used to separately store this information alongside the actual mesh data is created during the surface generation stage, as the Ptex parameters have to be partly computed during surface generation already. The per-primtive parameters include indices into the Ptex parameter buffer for all directly adjacent primitives, as well as its classification, that is which of the six cases of possible quads it maps to (see Figure 7). They also include indices into texture arrays, but these values will be set during texture baking and remain unitialized during the surface mesh reconstruction stage. By using the Naïve Surface Nets algorithm, computing the indices of the directly adjacent primitives is possible with minimal additional data, as vertices are shared among the generated quad primitives. For each generated vertex, we store the indices of generated quads (connected to this vertex) in a fixed pattern based on their classification and from which direction they connect to the vertex. For a given primitive, it is then possible to retrieve the indices of all neighbours based on the classification of the primitive and the shared edges, that is in what direction the primitives are connected. Apart from simply knowing a primitive's neighbourhood, rendering a Ptex mesh will require to transform between the local uv spaces of adjacent primitives. The total number of possible transformations between local uv-spaces of two adjacent primitives is limited to 16 and these can be stored in a lookup table [23]. The total number of possible combinations of classifications for two adjacent primitives generated by the Naïve Surface Nets algorithm is also limited, and as the local uv space of each classification is known, we can map each combination of classifications to one of the sixteen uv-space configurations. By storing this mapping in a lookup table as well, we can transform between the local uv spaces of adjacent primitives by storing only the classifications.

4.7 Texture Baking

Unlike a heightmap-based landscape, our dynamically created terrain surface mesh can have arbitrary shape. It thus offers no reasonable implicit mapping of the surface to 2D texture space. One way to texture such a mesh would be to use a semi-procedural approach to set the surface materials for the terrain and use tri-planar projection of tiling textures. This is also the approach we used in our previous work [1]. However, this technique offers no local control of the surface material and the shader program used for rendering becomes increasingly convoluted and costly by the introduction of multiple material layers for different altitudes, slope angles and other criteria. To remedy these shortcomings, the introduction of a material ID parameter to our Feature Curves and terrain generation pipeline enables local material control. For each mesh primitive, the different materials given by the IDs embedded in vertices are evaluated and blended within the primitive, offering local material control at the granularity of the surface mesh density. Nevertheless the complexity of evaluating the tri-planar projection of a multi-layered physically-based material for each triangle or quad vertex remains high and the rendering of the terrain surface costly. We also want to allow locally unique, high-resolution details instead of being limited to the surface mesh resolution. Therefore, we opted for completely decoupling the complex texture generation from the surface rendering by baking the result of a tri-planar projection and material blending into textures and only perform simple texture fetches during rendering. Obviously, baking the textures in advance requires some sort of surface parametrization for generating texture coordinates. We solve this issue by using a real-time implementation of per-face texture mapping (Ptex), introduced by Burley and Lacewell [24], which is widely adopted in the CG industry for offline production renderers and content generation. Our implementation of Ptex for real-time rendering is based on the presentation by John McDonald [23]. The basic idea of Ptex is that each primitive of a mesh is assigned its own texture using a local uv-space that is simply given by the vertex order of the primitive. Consequently, Ptex requires no explicit uv-mapping for the mesh and is thus a perfect fit for our terrain mesh.

In the texture baking stage, we first create texture arrays to store the individual Ptex textures for the quad primitives of the surface mesh. Each group of four consecutive layers in a texture array is assigned to a single quad and stores its albedo, specular, roughness and displacement texture maps as well as a normal map. We also refer to the single layers of the texture arrays as (Ptex) texture tiles. As the number of layers used for a texture array is limited by the hardware, we may need hundreds of texture arrays. Using OpenGL's Bindless Texture feature makes the texture arrays easily accessible in our shader programs via a GPU buffer

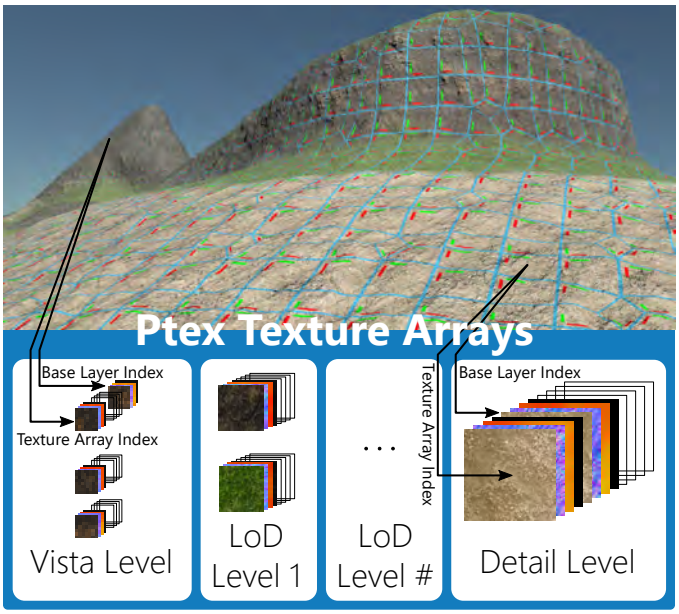

Fig. 8. Screenshot of terrain mesh using Ptex with texture tile borders and local uv-spaces baked into the texture tiles and overview of Ptex texture arrays in our level of details texturing solution.

that contains all bindless texture handles. The per-primtive Ptex parameters introduced in section 4.6 contain an index into this texture handle buffer as well as a base layer index to access the group of texture tiles that are associated with a primitive. Since we aim to have a maximum texture tile resolution of 256×256, storing the Ptex textures for all quads in full resolution with a complete mip-map chain in GPU or main memory at all times is simply not feasible. A medium-sized landscape as shown in figure 10 already consists of 16970 primitives and theoretically requires more than 18GB of texture storage. Using texture compression lowers the memory consumption but obviously does not scale and adds further complexity as we cannot easily bake our texture data directly into compressed texture memory. However, the highest texture resolution is only required for quads close to the viewer with an increasing amount of lower resolution tiles required by quads farther away. Therefore, we employ a hierarchical texturing solution with several detail levels that contain texture tile resolutions ranging from 8×8 to 256×256. This strictly limits the number of high resolution tiles in order stay within a user-defined memory budget for all Ptex texture. Figure 8 shows a terrain mesh using Ptex with texture tile borders and local uv-spaces baked into the texture tiles. It also illustrates the usage of Ptex texture arrays from different detail levels. A basic vista level made from 8×8 texture tiles serves both as the display texture for far away quad primitives and as a fallback texture when no higher resolution tile is available yet. It contains texture tiles for all primitives and is present in GPU memory at all times. The remaining Ptex texture memory is equally split among the levels with higher resolution. Note that higher resolution levels require more memory for covering the same amount of primitives, which automatically reduces the amount of available texture tiles per resolution level. After the texture arrays for all detail levels are created, the vista layer is baked, that is, the actual texture information is written to its texture tiles.

Texture baking is done per primitive and essentially uses the same method that was previously used during rendering and has been discussed above. It uses the material information from the primitive vertices and blends the result of the tri-planar texture projection of these materials. Beforehand, it also evaluates a noise function based on the noise parameters embedded in the primitive vertices. The noise values are used as surface displacement and are combined with displacement map of the material and then written to the Ptex texture tiles. Note that changes of the surface normals due to the noise-based displacement have to be considered for the tri-planar projection. In addition to generating the texture data using the tri-planar projection of tiling material textures, Ptex also allows to use decals that can be directly baked into the texture tiles. By baking the decals into the textures, no additional overhead is added to rendering. With this strategy it becomes possible to cover up visible seams and repeating patterns in the tiling textures or to simply decorate the landscape with small, high-resolution details such as small craters or puddles. Finally, the texture tiles of the remaining detail levels are baked.

As the amount of texture tiles in the non-vista levels is limited, they have to be carefully distributed among the surface primitives that are actually visible, that is, among the primitives within the camera's viewing frustum. While the initial computation for these levels is done in the generation pipeline's final stage, the non-vista-level texture tiles are continuously reassigned and updated accordingly when the camera view and the primitives within the frustum change. The first step of such an update involves checking each primitive against the camera's frustum and computing the distance to the camera. These distance values are then sorted and each primitive is classified into a detail level based on its distance and the overall number of texture tiles per detail levels, with closer primitives assigned to higher levels. Once the distance values pass certain thresholds, primitives are classified into lower detail levels even if the higher levels are not completely used up yet. By comparing a newly made classification to the last known classification, we identify primitives that need to be updated and store them in a list. Primitives that move from a non-vista layer also contribute their previously assigned texture tiles to a list of free texture slots per detail level. From these two lists, we can then compute the new Ptex texture tile assignment and bake the texture data of reassigned texture tiles.

5 Optional Extensions of our Method

To improve usability and scalability of our approach, we also implemented two extensions that are described in the following.

5.1 Feature Curve Expansion

As an additional, optional stage in the pipeline it is possible to automatically expand the Feature Curves along the guidance fields $\mathcal{N}$ and $\mathcal{B}$. This decreases the open space not covered by any user-defined Feature Curves and potentially reduces the ambiguity of the terrain surface in some areas. A Feature Curve is expanded by making small steps in

the direction of the cross product of normal and bitangent vector. After some distance is covered, a new control vertex is added to the curve and the process is stopped once we either step out of the computational domain or get too close to another Feature Curve. After all Feature Curves have been expanded, the guidance field $\mathcal{N}$ is updated to match the expanded Feature Curves.

To achieve a stable expansion, a densely populated, smooth bitangent field is required. We consider the bitangent vector field to be smooth if the dot product of adjacent vectors is either close to one or close to minus one, meaning that the sign is ignored. A diffusion process as used for $\mathcal{N}$, which basically computes the mean vale of a local neighbourhood, is not applicable. It is possible to formulate the computation of the most suitable bitangent vector for any given voxel based on a local neighbourhood as an optimization problem. However, to avoid a costly computation and a possibly more complicated implementation, a more straightforward function to compute a suitable bitangent vector is desirable. We settled on approximating a possible solution for a given voxel by several successive, pair-wise averaging operations using its local neighbourhood. As our averaging operation is neither commutative nor associative, changing the order in which vectors are averaged potentially changes the results. Thus, $\mathcal{B}$ is biased in some direction.

5.2 Large Terrain using Volume Bricking

So far the computational domain is limited to a single grid of rectangular shape, treating all areas within the domain uniformly. While this allows for clear definitions and facilitates the understanding of the terrain generation pipeline's core structure, in practice it leads to a potential waste of processing power. Regions where the terrain surface is not present at all or that are far away from the virtual camera do not need to be computed at full detail, that is at full grid resolution. Especially by taking advantage of adaptive resolution for distant regions, overall larger terrain becomes feasible.

To increase the flexibility of our approach, the domain can be subdivided into sub-regions called bricks. Just like the whole domain before, the individual bricks are rectangular uniform grids and each brick comes with its own set of fields $\mathcal{N}$, $\mathcal{B}$, $\mathcal{R}$, and $\mathcal{S}$. However, the resolution of the individual bricks can vary and the resolution of the domain as a whole is thus no longer uniform. As the computations of the terrain generation pipeline are now distributed over multiple bricks, global solutions (*w.r.t.* the whole domain) for the guidance field diffusion, the noise parameter diffusion, and the surface propagation need to be computed. To achieve this, the bricks have to exchange values with adjacent bricks at the shared boundaries, meaning that the boundary conditions of computations have to be modified to access values from adjacent bricks. Furthermore, the exchanged values have to be updated after each iteration, requiring synchronisation between bricks. Another issue is the reconstruction of the actual surface mesh. If the mesh reconstruction is simply performed for all bricks individually, a separate terrain surface mesh is constructed for each brick. Due to the properties of the mesh reconstruction algorithm, a small, empty gap remains between adjacent bricks that needs to be closed by inserting additional geometry.

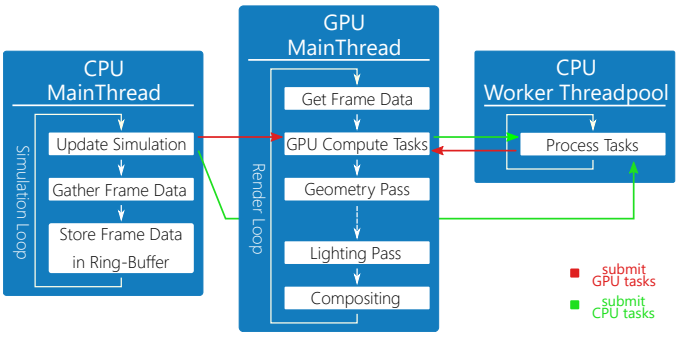

Fig. 9. A simplified overview of how simulation (or game-logic) and rendering are handled in the engine and interact with each other.

6 Implementation of Application Prototype

As a proof of concept of the presented terrain generation method, we implemented a prototype application. It is written in C++ and heavily relies on OpenGL (4.5) for general purpose computations on the GPU (GPGPU) as well as for the graphical presentation. Figure 9 shows a simplified overview of how computations and rendering are handled in the engine. The time budget for GPU compute tasks executed in a frame is limited to avoid stalling the rendering. All stages of the terrain generation pipeline use GPU acceleration to achieve interactive frame rates during editing of the terrain.

To correctly average over all intersecting triangles, the voxelization stage is split into a gathering and averaging step. First, the triangle intersection tests are performed in parallel and the results are gathered in a per-voxel linked list using the Shader Storage Buffer and Atomic Counter features [25]. In the averaging steps, the information stored per voxel is then averaged correctly. In each iteration of the iterative algorithms used in the normal field diffusion and surface propagation stage, the result for all voxels is computed in parallel, while the iterations itself are performed sequentially to allow synchronisation between iterations. The number of iterations depends on the grid resolution and should guarantee that information can propagate across the whole domain. The surface reconstruction uses a simple GPU implementation of the Naïve Surface Nets algorithm that uses atomic counters to tightly pack vertex and index buffer data in GPU memory. We can export the final surface mesh including vertex displacement using Transform Feedback.

Baking of the Ptex texture data is also mostly done on the GPU using compute shaders. For updating the Ptex texture tiles on demand, the distance computation and camera frustum check is performed in each frame on the GPU. This data is then sent to the CPU for sorting, primitive classification and comparison with the result from the previous frame. The primitive update lists are generated on the CPU as well and are then transferred back to the GPU for the actual texture baking. Note that the CPU tasks are performed completely asynchronously and do not stall the rendering thread. To capture the full surface details given by the texture data, we use adaptive surface tessellation using the baked displacement values.

The application offers a basic user interface to freely

Fig. 10. Three stages of modelling a simple oceanic island. From left to right: Start by modelling the island's shoreline. Then add Feature Curves to define the island's topology, in this case a small mountain and cliff. Finally, add some details to the major terrain features by placing additional Feature Curves.

create terrain surfaces and edit all properties included in our approach. The workflow of our method is centred around the arrangement of Feature Curves and offers the following functionality:

- Add and insert control vertices and Constraint Points anywhere on a curve
- Move control vertices and manipulate bitangents directly in the 3D viewport
- Switch between real-time and on-demand updates of the generated terrain
- Import and export Feature Curves as well as import heightmaps and meshes and export the final terrain surface mesh

7 Results and Discussion

In this section, we analyze our method with regard to the capabilities and the quality of the resulting terrain surface. We will also show examples of more complex landscapes that demonstrate the benefits of our method with regard to terrain features that cannot be represented by heightmaps. Furthermore, we evaluate the performance of our terrain generation pipeline.

7.1 Capabilities

Figure 10 shows how a simple oceanic island can be modelled in three steps. The images were captured directly from our prototype application and a grid resolution of $128 \times 64 \times 128$ was used for the scene. We started with a single Feature Curve that defines the shoreline of the island. We then modelled the major ridge line of a small mountain with a single Feature Curve and used two more curves to define the base of the mountain. Using another Feature Curve, we added a cliff to the other side of the island. In this case we did not need a second curve to model the bottom of the cliff (as seen in Figure 1), because the curve is close enough to the shoreline and our method automatically bridges the gap. Finally, we added some more details to the mountain and cliff with a few additional Feature Curves.

Apart from the slightly overhanging cliff, the oceanic island makes little use of the key advantage of our method. To better demonstrate the capabilities of our method, we recreated the famous Double Arch located in Arches National Park, USA. The result is shown on the left in Figure 13. We used our application to model the scene and then imported the generated terrain surface mesh into Blender [26] for shading and rendering. Only 18 Feature Curves were necessary to model the scene. For the terrain

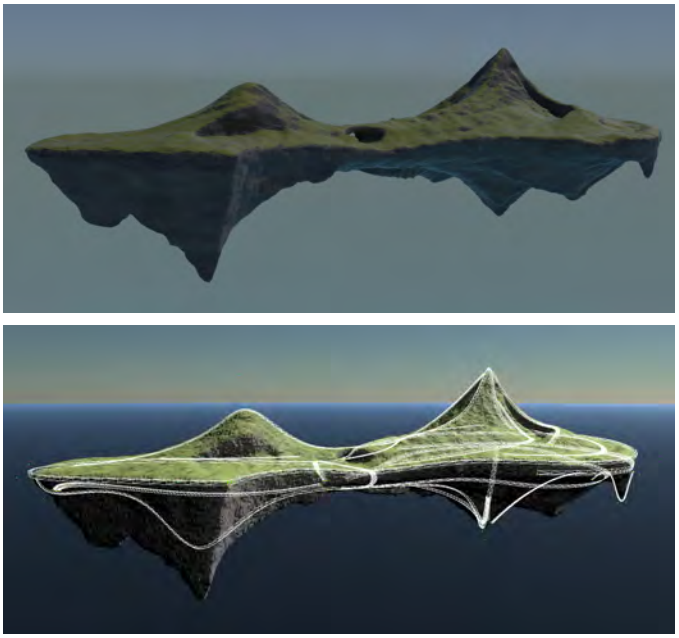

Fig. 11. Top: A fictional landscape consisting of floating islands. Our methods allows to consistently model both the topside and bottom of the islands using a single multi-purpose toolset. Bottom: A screenshot of the floating island in our prototype application showing the Feature Curves used to create the terrain surface.

generation, a grid resolution of $256 \times 64 \times 256$ was used. While our method has some difficulties in recreating small surface details and especially jagged rocks and cracks, it easily allows to model the primary arches and vertical rock faces. Overall, we observed that our methods tends to create smooth surfaces, due to the diffusion process used to create the normal vector field.

We also demonstrate the possibilities our method offers for the creation of fictional landscapes. The upper image in Figure 11 shows an island floating in the sky. This scene was also first modelled in our prototype application and then exported to Blender for rendering. We again used a grid resolution of $256 \times 64 \times 256$ and arranged 16 Feature Curves to cover all possible angles of the island. We also deliberately left some open space to have it automatically filled in by our method. This often results in organic and interesting shapes, as can be seen in Figure 11. On the bottom of the left half of the floating island, the geometry matching the Feature Curve is hidden behind geometry that was generated due the region being relatively unconstrained.

The three examples showcase that our method can be

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2017.2762304, IEEE Transactions on Visualization and Computer Graphics

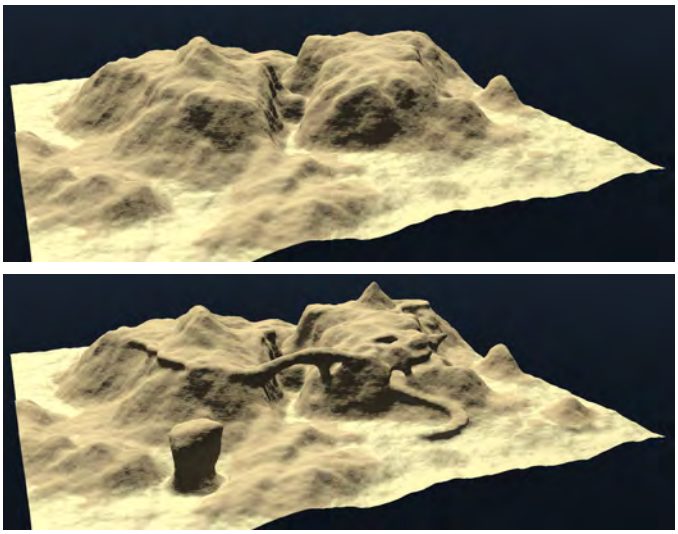

Fig. 12. Top: Terrain surface generated by our pipeline from only a heightmap input. Bottom: The heightmap base-layer is decorated using input meshes and Feature Curves. Feature Curves are used to add an arch, a path that winds up the mountain forming two natural underpasses and to make edges of already existing cliffs overhanging. Input meshes are used to add small hills and a large boulder.

used create complex, high-quality terrains with only few Feature Curves. During content creation, Feature Curves can be adapted intuitively to obtain a desired shape of the landscape or new Feature Curves can be added to introduce additional details. As the Double Arches example shows, our method can also be used to easily recreate existing landscapes (e.g. using a photograph or a drawing as template). In contrast to traditional heightfield-based terrain generation, our method allows to model layered structures like arches, caves or overhanging cliffs, which require a volumetric representation.

The ability to import terrain heightmaps and meshed terrain geometry adds value to our method, especially with regard to an actual production setting. That is, the output of existing tools and production pipelines can be imported into our framework and be freely combined with volumetric structures. A heightmap or input mesh can provide the base-layer of a landscape, which is then decorated using additional input meshes and Feature Curves, as shown in Figure 12. The example demonstrates that our Feature Curves are an effective tool for both adding new overhanging terrain features to a heightmap, but also to reinforce existing features that can not be represented well in a heightmap, e.g. the edges of steep cliffs.

The scene in Figure 14 showcases the texturing of the terrain controlled by the Feature Curve material parameters. By using Ptex, we can easily and cheaply add decals to the terrain surface texture. Decals are useful to add small details, mask the obvious transitions between materials, and to cover up the repeating patterns of tiling textures.

7.2 Performance

The overall performance of the terrain generation almost exclusively depends on the grid resolution. The number of Feature Curves only impacts the performance of the voxelization stage, while the complexity of the generated terrain has a minor impact on the performance of the surface reconstruction stage. The increase in execution time with increasing grid resolution matches theoretical expectations. Doubling the grid resolution in all dimensions increases the size of the computational domain by a factor of 8. Additionally, the numbers of iterations performed in the guidance field diffusion, noise field diffusion, and surface propagation has to be increased by a factor of two. Overall, this yields an expected increase in execution time by a factor of 16, which can also be observed in the performance measurements shown in Table 1. Our results show that a grid of size $128 \times 64 \times 128$ can still be computed in about one eighth of a second with a NVIDIA GTX1080, but for larger grids, additional optimizations will be necessary to reach interactive frame rates during terrain editing.

We also measured the performance of dynamically updating Ptex texture tiles and compared the rendering performance of the Ptex texturing solution against our previous one. For the view seen in Figure 8, the geometry pass of the render pipeline requires an average of 2.8 ms when using Ptex and up to 10 ms using our legacy renderer. Measuring the average time it takes to update the Ptex textures is difficult as it largely depends on the speed of camera movement. Table 2 shows the average and maximum time required by a texture update for two scenarios we benchmarked, as well as the average and maximum amount of primitives that were updated. Both scenarios are located in the teaser image landscape. It consists of 121,696 quad primitives and the memory requirements for Ptex textures are 125 MB for vista texture arrays and 537 MB for the detail texture tiles. A memory budget of at least 1 GB for Ptex textures is therefore recommended. Our first

TABLE 1
Computation times of the individual stages of the terrain generation pipeline at different grid resolutions, as well as for the complete pipeline. All timings were measured on an NVIDIA GTX1080.

Pipeline stage	$64 \times 32 \times 64$	$128 \times 64 \times 128$	$256 \times 128 \times 256$
Reset	0.03 ms	0.71 ms	5.04 ms
Voxelization	0.14 ms	0.50 ms	0.45 ms
Guidance field diffusion	3.47 ms	54.81 ms	570.04 ms
Noise field diffusion	2.21 ms	26.36 ms	377.52 ms
Surface propagation	3.21 ms	36.74 ms	473.07 ms
Surface reconstruction	5.87 ms	8.77 ms	20.13 ms
Texture baking	2.06 ms	3.85 ms	34.57 ms
Complete pipeline	16.99 ms	131.69 ms	1,480.82 ms

TABLE 2
Average and maximum computation times and amount of updated primitives for a Ptex texture tiles update. Measurements are given for two scenarios with different camera movement patterns.

Update Step	Scenario 1 Fly-through	Scenario 2 Terrain Editing
Distance Computation	6.36 ms	6.35 ms
Classification *(avg/max)*	3.62 / 11.45 ms	2.96 / 10.98 ms
Ptex Tiles Baking *(avg/max)*	1.04 / 2.13 ms	0.63 / 6.75 ms
Complete Update *(avg/max)*	11.02 / 19.94 ms	9.94 / 24.08 ms
Updated Primitives *(avg/max)*	508 / 2232	374 / 16726

Fig. 13. Left: A recreation of the Double Arch rock formation in Arches National Park, USA. The geometry was modelled in our prototype application and then imported into Blender for shading and rendering. Right: The photograph used as a template for the modelling. Photograph 'Rock Arch 1' courtesy of Fred Moore.

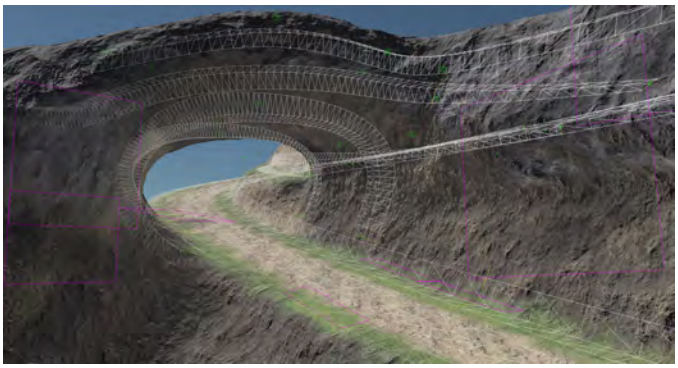

Fig. 14. Using Feature Curves to model and texture a narrow mountain path. Decals are used for decoration and to mask material transitions.

benchmark scenario is a fly-through that features smooth camera movements with a steady speed, resulting in a stable but moderately high amount of updated primitives per update. The second benchmark scenario measures the performance during terrain editing with our Feature Curve toolset. It features sudden camera movements, including fast 180° turns, but also a lot of minor camera adjustments. Keep in mind that a Ptex texture update is split into several independent steps, which are interleaved with rendering and partly performed asynchronously. Thus, the impact of an update on the frame time is limited and stuttering is avoided. However, fast camera movements can cause the texture updates to lag behind and low resolution tiles enter the field of view for a few frames.

8 Conclusion and Future Work

We proposed a novel approach to modelling and generating almost arbitrary terrain surfaces from sparse user input data, specifically aiming to enable the creation of terrain with arbitrary vertical layout, such as arches, caves, or overhanging cliffs. Our method combines the advantages of a volumetric terrain representation with the ability to quickly and efficiently edit large-scale terrain features by means of Feature Curves, a modelling tool based on three-dimensional parametric curves.

We first explained how parametric curves are outfitted with the necessary information for defining a terrain surface. Parametric curves are already used in the field of terrain modelling, however, we extended the existing approached by applying them to a volumetric terrain representation. Our proposed computational pipeline operates on a three-dimensional domain and combines voxelization, diffusion, and iso-surface extraction algorithms to generated a terrain surface from a set of Feature Curves. As the voxelization front-end of our pipeline is generalised to triangle meshes, the pipeline is also capable of importing heightmaps and meshes from other applications (e. g. meshed assets like rocks) and use them as additional terrain modelling primitives. We explained optional extensions that we added to the pipeline, including how to handle distributed computations in a subdivided domain for the purpose of improved flexibility and ultimately larger computational domains. The results showed that our method is capable of recreating complex terrain features that slope beyond the vertical, including overhanging rock faces, arches and the underbelly of a fictional floating island. Generating larger landscapes with many Feature Curves and a combination of numerous different terrain features is possible and we believe that curve-based modelling tools are suitable for modelling large-scale terrain features. By also controlling the surface material with the Feature Curves and using per-face texture mapping for the dynamically generated terrain mesh, texturing becomes more flexible and efficient.

Furthermore, we showed that our method is not only an effective standalone tool, but can be used to decorate existing heighmaps and terrain meshes. Finally, our GPU-accelerated application prototype allows for interactive editing of reasonably sized terrain regions, making it a suitable method for modelling terrain in real-time applications such as game engine level editors.

Several aspects could be improved in future work. To reinforce Feature Curves as a multi-purpose terrain modelling tool, additional constraint properties could be added, including for example vegetation density or more complex noise parameters (e.g. aimed at geologically motivated models). The normal field diffusion and surface propagation would benefit from further basic research into more accurate models of the local terrain surface contour. Finally, the performance of the computationally intense diffusion process used in the terrain generation pipeline could be

improved, for example by utilizing a multigrid method. Such an implementation could furthermore be incorporated into a coarse-to-fine level-of-detail scheme for the terrain using volume bricking.

ACKNOWLEDGMENTS

This work was partially fundend by Deutsche Forschungsgemeinschaft (DFG) as part of SFB 1244. We would like to thank the Free PBR Materials website (freepbr.com) for proving the textures used in the paper.

REFERENCES

[1] M. Becher, M. Krone, G. Reina, and T. Ertl, "Feature-based volumetric terrain generation," in *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, 2017, pp. 10:1–10:9.
[2] Hugh Honour, John Fleming, *A World History of Art*. Laurence King Publishing, 1995.
[3] L. C. Carpenter, "Computer rendering of fractal curves and surfaces," *SIGGRAPH Comput. Graph.*, vol. 14, no. 3, pp. 109–, 1980.
[4] Epic Games, Inc. (2016) Unreal engine 4 landscape outdoor terrain. https://docs.unrealengine.com/latest/INT/Engine/Landscape/index.html.
[5] Crytek. (2016) Cryengine terrain editor. http://docs.cryengine.com/display/CEMANUAL/Terrain+Editor.
[6] Unity Technologies. (2016) Unity manual - terrain engine. https://docs.unity3d.com/Manual/script-Terrain.html.
[7] M. Natali, E. M. Lidal, J. Parulek, I. Viola, and D. Patel, "Modeling terrains and subsurface geology," *Proc. EuroGraphics 2013 STARs*, pp. 155–173, 2013.
[8] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley, *Texturing and Modeling: A Procedural Approach*, 3rd ed. Morgan Kaufmann Publishers Inc., 2002.
[9] R. M. Smelik, K. J. De Kraker, T. Tutenel, R. Bidarra, and S. A. Groenewegen, "A survey of procedural methods for terrain modelling," in *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*, 2009, pp. 25–34.
[10] O. Šťava, B. Beneš, M. Brisbin, and J. Křivánek, "Interactive terrain modeling using hydraulic erosion," in *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA '08, 2008, pp. 201–210.
[11] J.-D. Génevaux, E. Galin, E. Guérin, A. Peytavie, and B. Benes, "Terrain generation using procedural models based on hydrology," *ACM Trans. Graph.*, vol. 32, no. 4, pp. 143:1–143:13, 2013.
[12] J. Gain, P. Marais, and W. Straßer, "Terrain sketching," in *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2009, pp. 31–38.
[13] F. P. Tasse, A. Emilien, M.-P. Cani, S. Hahmann, and N. Dodgson, "Feature-based terrain editing from complex sketches," *Comput. Graph.*, vol. 45, no. C, pp. 101–115, 2014.
[14] H. Hnaidi, E. Gurin, S. Akkouche, A. Peytavie, and E. Galin, "Feature based terrain generation using diffusion equation," *Computer Graphics Forum (Proceedings of Pacific Graphics)*, vol. 29, no. 7, pp. 2179–2186, 2010.
[15] A. Emilien, P. Poulin, M.-P. Cani, and U. Vimont, "Interactive Procedural Modelling of Coherent Waterfall Scenes," *Computer Graphics Forum*, pp. 1 – 15, 2014.
[16] A. Peytavie, E. Galin, J. Grosjean, and S. Merillou, "Arches: a framework for modeling complex terrains," *Computer Graphics Forum*, vol. 28, no. 2, pp. 457–467, 2009.
[17] K. Singh and E. Fiume, "Wires: A geometric deformation technique," in *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '98. New York, NY, USA: ACM, 1998, pp. 405–414.
[18] C. Xu and J. L. Prince, "Gradient vector flow: A new external force for snakes," in *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*. IEEE, 1997, pp. 66–71.
[19] P. Perona and J. Malik, "Scale-space and edge detection using anisotropic diffusion," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 12, no. 7, pp. 629–639, 1990.
[20] L. C. Evans, *Partial Differential Equations*. American Mathematical Society, 1998.
[21] W. E. Lorensen and H. E. Cline, "Marching Cubes: A high resolution 3d surface construction algorithm," in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '87. ACM, 1987, pp. 163–169.
[22] S. F. F. Gibson, "Constrained elastic surface nets: Generating smooth surfaces from binary segmented data," in *Proceedings of Medical Image Computing and Computer-Assisted Intervention*, ser. MICCAI '98. Springer-Verlag, 1998, pp. 888–898.
[23] J. McDonald, "Eliminating texture waste: Borderless realtime ptex," 2013, presented at GDC. [Online]. Available: https://developer.nvidia.com/gdc-2013
[24] B. Burley and D. Lacewell, "Ptex: Per-face texture mapping for production rendering," in *Eurographics Symposium on Rendering 2008*, 2008, pp. 1155–1164.
[25] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz, "Real-time concurrent linked list construction on the gpu," in *Computer Graphics Forum*, vol. 29, no. 4. Wiley Online Library, 2010, pp. 1297–1304.
[26] Blender Foundation, "Blender - open source 3d creation suite," 2016, http://www.blender.org.

Michael Becher received the degree in computer science from the University of Stuttgart, Germany in 2017. He is currently working towards a Ph.D. degree at the Visualization Research Center of the University of Stuttgart (VISUS). His research interests include real-time computer graphics, graphics-engine architecture, GPU-accelerated computing, and visual computing for structural engineering.

Michael Krone is a postdoctoral researcher at the Visualization Research Center of the University of Stuttgart (VISUS). He received his PhD computer science (Dr. rer. nat.) in 2015 from the University of Stuttgart, Germany. His main research interest include molecular visualization and visual analysis for structural biology, computer graphics, particle-based rendering, and GPU-accelerated computing.

Guido Reina is a postdoctoral researcher at the Visualization Research Center of the University of Stuttgart (VISUS). He received his PhD in computer science (Dr. rer. nat.) from the University of Stuttgart, Germany. His research interests include large displays, particle-based rendering, GPU-based methods, and fault-tolerant visualization.

Thomas Ertl is a full professor of computer science at the University of Stuttgart, Germany, and the head of the Visualization and Interactive Systems Institute (VIS) and the Visualization Research Center (VISUS). He received a MS in computer science from the University of Colorado at Boulder and a PhD in theoretical astrophysics from the University of Tübingen. His research interests include visualization, computer graphics, and human computer interaction.